UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Reingeniería de la herramienta de elaboración de mapas conceptuales con referencias (RCM) para el trabajo colaborativo

Autor/es

ÁLVARO CARRASCAL REPISO

Director/es

FRANCISCO JOSÉ GARCÍA IZQUIERDO y ÁNGEL LUIS RUBIO GARCÍA ,

Facultad

Escuela de Máster y Doctorado de la Universidad de La Rioja

Titulación

Máster Universitario en Tecnologías Informáticas

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2016-17

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

Trabajo de Fin de Máster

Reingeniería de la herramienta de elaboración de mapas conceptuales con referencias (RCM) para el trabajo colaborativo

Autor:

Álvaro Carrascal Repiso

Tutor/es: Francisco José García Izquierdo
Ángel Luis Rubio García

MÁSTER:
Máster universitario en Tecnologías Informáticas

Escuela de Máster y Doctorado

AÑO ACADÉMICO: 2016/2017

Índice de contenido

Índice de figuras

Resumen

Se dispone de una herramienta informática para la elaboración de mapas conceptuales con referencias (*RCMs*) a la que se pretende dar soporte para la edición colaborativa en tiempo real. Con este fin, ha sido necesario revisar el estado del desarrollo inicial de la herramienta para definir un patrón de arquitectura *MVC*, reestructurar las capas de la aplicación, ajustar la gestión interna de referencias bibliográficas y redefinir el modelo de datos existente. Se han incorporado varias características nuevas para subsanar las limitaciones de la versión inicial, realizándose mejoras en la visualización, mejorando el manejo de errores, actualizando los recursos existentes y añadiendo algunas funcionalidades avanzadas para mejorar las prácticas de la herramienta. De cara a simplificar las labores de importación de datos, se han sentado las bases para la integración con gestores de referencias externos. El resultado es una aplicación mucho más escalable, orientada al trabajo en equipo y a la compartición de recursos en tiempo real, con soporte para la integración con herramientas externas y mayor flexibilidad de cara a posibles desarrollos futuros.

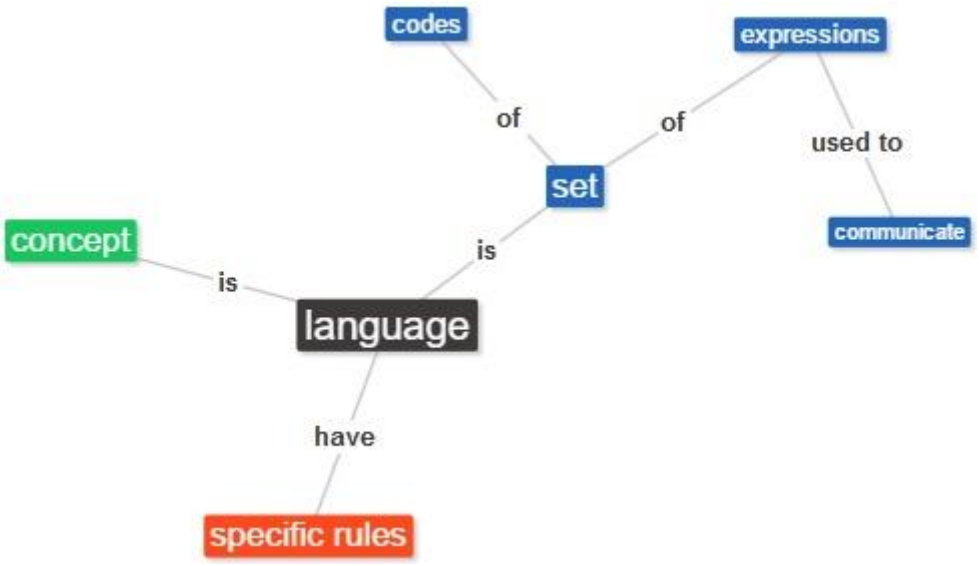
There is a computer tool for the development of conceptual maps with references (*RCMs*) which is intended to support real-time collaborative editing. For this purpose, it has been necessary to review the state of the initial development of the tool to define an *MVC* architecture pattern, reshape the application layers, adjust the internal management of bibliographic references and redefine the existing data model. Several new features have been incorporated to overcome the limitations of the initial version, making improvements in the visualization, improving the handling of errors, updating the existing resources and adding some advanced functionalities to improve the practices of the tool. In order to simplify the tasks of importing data, have laid the foundations for integration with external reference managers. The result is a much more scalable, team-oriented application and real-time resource sharing with support for integration with external tools and greater flexibility for future developments.

Introducción

"Cuando se realizan tareas de modelización, es importante que los distintos miembros de un equipo de trabajo compartan el mismo vocabulario. Esto no sólo implica un acuerdo sobre la terminología a utilizar, también sobre el significado de los términos utilizados. Surge, por tanto, la necesidad de utilizar herramientas gráficas capaces de gestionar el conocimiento en el uso de referencias múltiples. Fruto de esa necesidad nace el concepto RCM.

El concepto de RCM, o References-enriched Concept Map, hace referencia a una técnica basada en la utilización de mapas conceptuales que permite la visualización simultánea de un conjunto de definiciones sobre un término que han sido discutidas en la literatura, facilitando su análisis en relación con los autores que las propusieron." [1] [2]

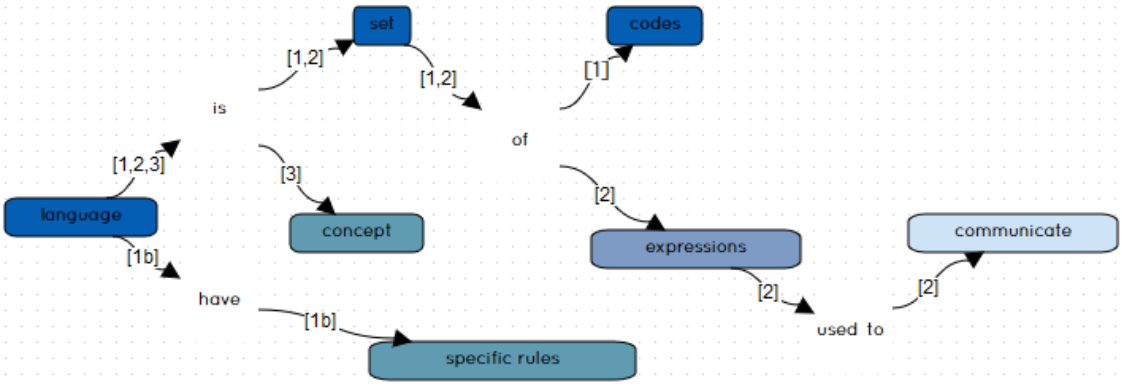
Un mapa conceptual [Figura 1] es una técnica utilizada para la representación gráfica del conocimiento que se compone por una red de elementos interconectados, o grafo, donde los nodos representan conceptos y los enlaces las relaciones entre ellos.



[Figura 1] Mapa conceptual

La técnica *RCM* toma como base esta técnica y añade la particularidad de que permite asociar una referencia bibliográfica a cada una de las definiciones reflejadas en el grafo mediante un etiquetado.

A continuación, se muestra un ejemplo ilustrativo para aclarar los conceptos, mostrando los distintos elementos que componen un *RCM* [Figura 2]. Obviamos por ahora la lista de referencias con la que trabaja el grafo.



[Figura 2] *RCM*

El *RCM* está formado por un grafo que contiene una serie de entidades que pueden ser de dos tipos: *Concepts* (entidades principales coloreadas con distintas tonalidades de fondo azul) y *LinkingPhrases* (entidades que hacen de nexo entre dos *Concepts*).

Un *RCM* permite etiquetar mediante *Labels* secuencias de *Concepts* y *LinkingPhrases*. El conjunto de *Concepts* y *LinkingPhrases* identificados mediante un mismo *Label* que permanece invariable a lo largo de varios niveles de profundidad del grafo recibe el nombre de *Path*. Cada *Path* de un *RCM* corresponde a una de las definiciones incluidas en ese *RCM*.

Un *Label* de un *Path* está formado por uno o más caracteres (se omite el segundo siempre que corresponde al carácter "*a*"). El primero de ellos indica el identificador numérico de la referencia bibliográfica (*Reference*) a la que pertenece (*1*, *2*, *3*, etc.), mientras que el segundo es un mero contador incremental alfabético ("*a*", "*b*", "*c*", etc.) que lo diferencia del resto de *Paths* que tienen asociada esa misma *Reference*. Si seguimos un determinado *Path* partiendo del *Concept* principal, comprobamos que determina una definición concreta incluida en el *RCM*. De esta forma, el *Path* permite relacionar una definición del *RCM* con una *Reference* concreta.

Más adelante, incidiremos de nuevo en los elementos que forman parte de un *RCM* y su tratamiento de la información.

Volviendo a la técnica *RCM* como mecanismo de representación del conocimiento, surge la necesidad de desarrollar una herramienta que dé soporte a esta metodología de trabajo. Como consecuencia se desarrolló *RCMTool* [3], una aplicación creada con el propósito de facilitar la usabilidad en los procesos de creación y análisis de *RCMs*.

La herramienta permite al usuario recopilar una serie de sentencias y le ofrece como resultado un grafo sobre el cual tendrá la posibilidad de añadir, modificar y eliminar elementos, además de editar sus propiedades. También ofrece mecanismos para la asociación entre las referencias y los elementos del grafo, a través de un etiquetado dinámico de relaciones o *Path Labeling*.

Contextualización y antecedentes

La técnica RCM nace de la mano de Emilio Rodríguez Priego, Francisco José García Izquierdo y Ángel Luis Rubio García, todos ellos pertenecientes al personal docente e investigador del Departamento de Matemáticas y Computación de la Universidad de La Rioja. Es a finales de 2013 cuando publican en conjunto una serie de artículos, con motivo de la tesis doctoral de Emilio Rodríguez Priego, en los que ponen de manifiesto la necesidad de implementar una herramienta específica de trabajo para la agrupación y comparación de un conjunto disperso de definiciones que aparecen relacionadas en torno a múltiples referencias.

En la publicación del 6 de diciembre de 2013 de la *Journal of Information Science*, una revista dedicada a la investigación sobre la ciencia y gestión de la información y otras ramas del conocimiento, encontramos el artículo que sienta las bases del proyecto *RCMTool* [2]. En él se recoge, por primera vez, la definición del concepto *RCM*, se publican las bases para su construcción y se establecen una serie de métricas para su análisis. También detalla una relación de pautas a seguir para la implementación de una herramienta capaz de dar expresividad a la técnica.

Ya en el año 2014, Carlos Sáenz Adán, actual investigador y estudiante de doctorado de la Universidad de La Rioja, desarrolla una primera versión de la

herramienta *RCMTool* como resultado de su tesis de *Máster en Sistemas Informáticos Avanzados* [1] impartido por la Universidad del País Vasco.

Podemos dividir su trabajo en dos enfoques claramente diferenciados en función del área del conocimiento al que afectan: el primero de ellos desde un ámbito semántico/lingüístico, mientras que también hay una parte del proyecto centrada en un punto de vista más técnico.

Empezando por el enfoque lingüístico, se centra en la descripción de una serie de fundamentos que permitan asentar una base que restrinja de algún modo el uso de la filosofía *RCM* y sobre la que sostener la implementación de futuras funcionalidades. Con el fin de dar expresividad a la técnica, define un metamodelo que permita evaluar si la estructura de un grafo es válida respecto a la estructura general de un *RCM*, lo que supone establecer una serie de restricciones para la construcción de un *RCM*, limitando las condiciones que debe cumplir un grafo para ser considerado válido.

En el marco del procesamiento del lenguaje, se centra en la implementación de un algoritmo basado en el etiquetado de referencias que permite mejorar la usabilidad de la herramienta, aportando el valor añadido de que el programa sea capaz de procesar una definición dada por el usuario y esta quede reflejada dinámicamente sobre el grafo de trabajo en tiempo real.

Por otra parte, desde la perspectiva técnica, más cercana a lo que acontece respecto a este TFM, fija una serie de requisitos funcionales que se le deben presuponer a la herramienta. Los que nos interesan y que deben tratarse para la comprensión del método de trabajo tienen que ver con la gestión de *References* y su interacción con el grafo para permitir el etiquetado dinámico, o P*ath Labeling*, a través de *Paths*.

El resultado es una aplicación basada en una arquitectura web cuyo contenido está alojado en un servidor y es accesible a los usuarios a través de internet mediante un navegador.

Fundamentos tecnológicos y estado inicial

RCMTool es una aplicación escrita en su totalidad en lenguaje *JavaScript*. Utiliza el intérprete *Node.js* que permite construir aplicaciones altamente escalables, manejando un gran número de conexiones simultáneas desde una única máquina física.

Node.js [4] es un entorno de ejecución de E/S dirigido por eventos, y por lo tanto asíncrono, que se ejecuta sobre el intérprete de *JavaScript* V8 creado por *Google* para su navegador *Chrome*. No es el único entorno de ejecución de este tipo, pero sí el primero basado en *JavaScript* y que ofrece un gran rendimiento. Aprovechando el motor V8 proporciona un entorno de ejecución del lado del servidor que compila y ejecuta *JavaScript* en código máquina nativo, lo que le permite alcanzar velocidades muy altas. Además, *JavaScript* tiene la ventaja de poseer un excelente modelo de eventos, ideal para la programación asíncrona.

Node.js cuenta con un manejador de paquetes por defecto, *npm* o *Node Package Manager*, que es instalado automáticamente con el entorno. *Npm* se ejecuta desde línea de comandos y maneja las dependencias de las aplicaciones mediante un fichero de configuración, *package.json,* que contiene un resumen del versionado de las dependencias de un proyecto. Además, *npm* permite a los usuarios instalar aplicaciones *Node.js* que se encuentran en su repositorio oficial.

Node.js se sirve de *Express.js* [5] como *framework* de aplicaciones web. Incorpora una infraestructura mínima y flexible que permite el manejo de servidores *HTTP* y provee de *plug-ins* de alto rendimiento conocidos como middleware. *Express.js* representa una arquitectura de software que modela las relaciones generales de las entidades de dominio, y provee una estructura y una metodología de trabajo propios.

El patrón de arquitectura propuesto es el *MVC*, o Modelo-Vista-Controlador. Se trata de un modelo organizativo que separa los datos de una aplicación, la interfaz de usuario y la lógica de control en tres componentes distintos. Es uno de los estilos de arquitectura de software más maduros y que ha demostrado su validez a lo largo de los años en todo tipo de aplicaciones, sobre multitud de lenguajes y plataformas de desarrollo.

En la capa de persistencia cuenta con un sistema de base de datos *NoSQL MongoDB* [6] orientado a documentos. La principal diferencia con respecto a las bases de datos relacionales es que, en lugar de guardar los datos como registros de una tabla, almacena los datos como documentos *BSON*, una representación binaria del formato *JSON*. *MongoDB* ofrece una gran agilidad y escalabilidad ya que no es necesario seguir una estructura definida y los documentos de una misma colección pueden seguir esquemas diferentes.

Para la capa de presentación utiliza *Hogan.js* [7]. *Hogan.js* es un motor de plantillas ligero desarrollado por *Twitter* que se integra a la perfección con *Node.js* y *Express.js*. Es posible usarlo como recurso para compilar plantillas o incluirlo en el navegador para administrar la generación de plantillas dinámicas. Es rápido y posee una *API* para análisis.

El conjunto de tecnologías de las que hemos hablado hasta el momento: *Node.js*, *Express.js*, *MongoDB* y *Hogan.js*, tienen en común dos características: son tecnologías que se utilizan en el lado del servidor, o *back-end*, y todas ellas han sido desarrolladas bajo el concepto de código abierto.

Nos centramos ahora en el lado del cliente, o *front-end*. El *framework jQuery* [8] proporciona una infraestructura que aporta mayor facilidad para la creación de aplicaciones complejas del lado del cliente: permite simplificar la manera de interactuar con los documentos *HTML*, manipular el árbol *DOM*, manejar eventos, desarrollar animaciones y agregar interacción a páginas web mediante la técnica *AJAX*. Se trata de una biblioteca multiplataforma de *JavaScript* libre y de código abierto.

El verdadero elemento de distinción que ofrece la herramienta y el principal motivo por el cual se desarrolló es ofrecer la capacidad de trabajar específicamente con *RCMs*. El elemento conceptual en torno al que se forma un *RCM* es el grafo. Es aquí donde entra en juego la tecnología *Rappid*.

Rappid [9] es un *framework* de diagramación para aplicaciones avanzadas, que ofrece lo mejor de las tecnologías *HTML5* y *SVG* y que proporciona las herramientas adecuadas para favorecer la creación de aplicaciones visualmente potentes y atractivas. Utiliza la librería *JavaScript JointJS* para visualizar y

administrar diagramas, formas y gráficos. Entre sus características, destaca que sigue el patrón *MVC* y ofrece soporte para *Node.js*. Se trata de una librería de código cerrado que ha sido cedida para la realización de este proyecto por considerarse con fines académicos.

El funcionamiento de la herramienta es bastante intuitivo. El acceso a la aplicación se realiza mediante un sistema de autenticación usuario-contraseña. Una vez autorizado, el usuario tiene la posibilidad de gestionar su propia lista de *RCMs*. Sobre ella tiene la posibilidad de crear, editar y eliminar elementos.

El verdadero potencial de la herramienta radica en la manejabilidad que ofrece para la edición de *RCMs*. El usuario tiene la posibilidad de añadir definiciones al grafo de forma dinámica, a través de sentencias sintácticamente bien construidas. *RCMTool* incorpora un metamodelo encargado de precompilar cada una de las definiciones para comprobar si la construcción de una sentencia es estructuralmente correcta, quedando restringidos todos aquellos casos de uso que no forman semánticamente una frase válida. También permite añadir nuevos elementos al grafo de forma manual, ofreciendo una mayor flexibilidad en el proceso de edición del grafo.

En líneas generales, la funcionalidad de la herramienta se ajusta tanto a los criterios teóricos definidos inicialmente por el documento publicado en la *Journal of Information Science*, como a los requisitos descritos con posterioridad en su estudio previo al desarrollo inicial. Sin embargo, al igual que ocurre con el resto de desarrollos de software, surge la necesidad de evolucionar, dotar a la herramienta de nuevas funcionalidades que añadan valor al producto y, en definitiva, facilitar al usuario una serie de mecanismos que le permitan alcanzar su objetivo de la manera más fácil y rápida posible.

Comprensión de conceptos generales de la herramienta

Para el seguimiento del documento de la memoria es necesario conocer una serie de conceptos que permitan al lector comprender la metodología de trabajo de la herramienta.

RCMTool dispone de un editor que permite al usuario generar y manipular su lista de *RCMs*. Dado que ya se ha definido anteriormente el término *RCM*, nos limitaremos a incidir en que es una técnica que se basa en el enriquecimiento de mapas conceptuales a través de los siguientes elementos: *References, Sentences* y *Paths*.

- *Reference*

Una *Reference*, o referencia bibliográfica, no es más que un conjunto mínimo de datos que permite la identificación de una publicación científica o de una parte de la misma.

El usuario tiene la posibilidad de añadir referencias a la aplicación a través de porciones de texto en formato *BibTeX*. *BibTeX* [10] se define como una herramienta que permite dar formato a listas de referencias. Facilita la realización de citas bibliográficas de un modo consistente mediante la separación de la información bibliográfica de la presentación de esta información. Este mismo principio de separación del contenido y presentación/estilo se usa por otros sistemas de formato electrónico de textos, como *LaTeX*.

- *Sentence*

Una *Sentence*, o sentencia, está formada por un conjunto de elementos que hacen mención a una definición bien formada del grafo, sintácticamente correcta y con sentido propio. Desde el punto de vista semántico, una sentencia está formada por dos o más *Concepts (un número n)*, unidos entre sí mediante *LinkingPhrases (n - 1)*, es decir, un *Path*.

- *Path*

Un Path representa una proposición existente dentro del *RCM*. Se trata de un elemento que permite relacionar una definición bien formada del grafo (*Sentence*) con la referencia bibliográfica (*Reference*) en la que se recoge dicha mención en la literatura.

Cada *Path* únicamente puede estar asociado a una *Reference*, mientras que una *Reference* puede estar ligada a varios *Paths*.

La metodología de trabajo permite al usuario facilitar una serie de definiciones a la herramienta y esta se encarga de generar el grafo correspondiente y ofrecérselo visualmente al usuario. Este puede añadir, modificar y eliminar elementos del grafo, además de establecer relaciones entre sus elementos.

El metamodelo definido está compuesto por entidades de tipo *Concept* y *LinkingPhrases*. La diferencia ontológica entre ambos elementos se determina a través de su significado en el mundo real. Mientras que los *Concepts* representan "cosas y eventos", los *LinkingPhrases* representan "actos y procesos".

- *Concept*

Recogen los términos más importantes del grafo. Tienen la particularidad de que se clasifican según el *Layer* al que pertenezcan.

- *LinkingPhrase*

Recogen aquellos términos que relacionan varios *Concepts* entre sí. La existencia de un *LinkingPhrase* está ligada a los Concepts, ya que carecen de sentido sin un elemento origen y un elemento destino.

Adicionalmente, se utilizan otro tipo de entidades y conceptos que, a pesar de no tener peso en la definición del metamodelo de los *RCM*, tienen cierta relevancia a la hora de entender la disposición de los elementos en el grafo.

- *Link*

Se trata de entidades que relacionan con *Concepts* y los *LinkingPhrases.* Establecen los vínculos existentes entre los elementos del grafo y son las estructuras encargadas de gestionar el *Path Labeling* o etiquetado.

- *Layer*

El *layering* se trata de una característica que permite resaltar la importancia de cada *concept* dentro del *RCM*.

Una determinada sentencia tiene una serie de *Concepts* asociados. En función del nivel de importancia del *concept* en el *RCM* se distinguen varios

niveles de clasificación (primario, secundario, terciario o cuaternario) asociados a una escala de tonos de color correlativos.

- *Path Labeling*

Se define como el proceso de etiquetado que permite identificar las referencias bibliográficas a las que está vinculado un determinado *Path* del *RCM*.

- *Syntagm*

Se trata de una entidad que facilita el ordenamiento de los *Concepts* y *LinkingPhrases* dentro de un mismo *Path*. Un elemento *Syntagm*, o sintagma, está compuesto por un *LinkingPhrase* y su *Concept* inmediatamente posterior dada una definición.

Objetivos

Una vez expuesto el marco de trabajo en el que se encuadra el TFM y los fundamentos teóricos en los que se basa, es necesario plantear sus motivaciones: qué se pretende y cuáles son los puntos de actuación sobre los que se pretende incidir.

A continuación, se enumeran los puntos objeto del proyecto teniendo en cuenta su nivel de importancia en la realización del proyecto:

- **Permitir la edición de *RCMs* de forma colaborativa en tiempo real**

  Se plantea como concepto prioritario y fundamental del proyecto conseguir que varios usuarios puedan trabajar simultáneamente sobre los elementos de un mismo *RCM* de forma transparente, haciendo a su vez partícipes a todos sus integrantes de los cambios producidos sobre el *RCM* en tiempo de ejecución. Cuando hablamos de trabajar sobre un *RCM* nos referimos a la posibilidad de que varios usuarios puedan realizar modificaciones en cualquiera de los elementos del editor: el grafo y/o sus listas de *References* y *Paths* asociadas.

  La herramienta debe enfocar su metodología de trabajo hacia un punto de vista colaborativo, donde prima el trabajo en equipo. Se deben proponer mecanismos que permitan, por un lado, la posibilidad de que un usuario pueda compartir sus *RCM* con el resto, y por otro, el acceso concurrente de varios usuarios sobre un mismo *RCM*. Surge la necesidad de controlar los accesos que se realizan en un determinado momento a un *RCM* y gestionar el nivel de privilegios que tiene cada usuario del sistema sobre cada uno ellos.

  Aprovechando la capacidad que ofrece *Node.js* para la gestión de peticiones asíncronas, será necesario realizar los cambios oportunos en el proyecto para facilitar a los clientes las consultas del estado de un *RCM* en un determinado momento. El servidor deberá ofrecer un servicio de suscripción de notificaciones que permitirá que cualquier modificación realizada en el grafo o en las listas de *References* y *Paths*, desencadene una serie de

eventos que habiliten la comunicación con los clientes, de tal forma que estos puedan reflejar los cambios producidos en tiempo real.

Hay que hacer especial hincapié en que, además de la complicación intrínseca del objetivo, su consecución conlleva una dificultad añadida ya que supone un cambio drástico en la filosofía de trabajo de la herramienta. En su versión inicial, la mayor parte de la lógica de negocio de la vista del editor se gestiona en el lado del cliente. Siguiendo esta metodología, se antoja complicada la sincronización de información entre dos clientes, los cuales no tienen ninguna vía de comunicación entre ellos más allá de la comunicación individual que puedan tener con el servidor. En cualquier caso, si la información se gestiona exclusivamente en el lado del cliente, cualquier cambio producido no puede ser propagado al resto. Para subsanar este problema se requiere trasladar toda la lógica de negocio al lado del servidor, lo que conlleva un cambio total de la dinámica de trabajo.

Adicionalmente, y a pesar de que no se trata de un objetivo principal, de esta forma conseguimos aligerar el peso de la aplicación en cliente, con lo que ganamos en fluidez y usabilidad a la hora de interactuar con el usuario.

- **Integrar la herramienta con el gestor de bibliografía *Mendeley* y sentar las bases para futuras integraciones con otros gestores de referencias**

  Uno de los puntos fuertes de la herramienta radica en la posibilidad de integración con gestores de referencias externos. Se propone la integración con *Mendeley*, uno de los gestores de referencias con mayor número de usuarios en su comunidad (en torno a 3 millones) y que dispone de una base de datos con más de 100 millones de referencias.

  Mendeley [11] permite gestionar y compartir referencias bibliográficas y documentos de investigación, encontrar nuevas referencias y documentos, y colaborar en línea. Dispone de una *API* pública que facilita el acceso a su información a través del protocolo de autorización *Oauth* [12] que permite el acceso de aplicaciones de terceros a contenidos propiedad del usuario sin

necesidad de que estas aplicaciones tengan que manejar ni conocer las credenciales de autenticación del usuario.

Aprovechando que se trata de la primera integración con un gestor de referencias externo, se deberán abstraer las generalidades del proceso de intercambio de información para facilitar la reutilización de los controladores desarrollados en futuras integraciones con otras *APIs* de aplicaciones similares.

- **Revisar la gestión de referencias**

  Uno de los puntos que se detecta susceptible de mejora tiene que ver con la forma en que *RCMTool* maneja internamente las referencias.

  Tal y como se plantea inicialmente, una referencia es una entidad que pertenece a un grafo y que se gestiona como si se tratase de un elemento más. Si una misma referencia se añade a más de un grafo, no existe relación aparente entre ellas, coexistirán en la base de datos tantas copias de la referencia como veces aparezca relacionada a un grafo.

  A partir de ahora, el concepto de referencia cambia. Es necesario tratar una referencia como una entidad única del modelo de datos e independiente del grafo.

  Por otra parte, surge la necesidad de identificar un conjunto de referencias concreto del resto. La probabilidad de que un mismo usuario trabaje con un conjunto determinado de referencias en más de un *RCM* es bastante frecuente. Nace así el concepto de carpeta: un elemento contenedor que permite gestionar un conjunto determinado de referencias como una entidad propia.

  Cada usuario debe ser capaz de gestionar su propia lista de referencias y, a su vez, trabajar con su propia lista de carpetas.

- **Dotar al proyecto de una estructura interna bien definida y acorde a futuras funcionalidades**

  Dada la naturaleza del tipo de máster en el que surge la aplicación inicial, cuyo propósito pretende cubrir ámbitos de la semántica desde un punto de vista informático, el desarrollo de la primera versión de la aplicación centra gran parte de su trabajo en sentar las bases teóricas de la herramienta. Es decir, intenta definir una metodología de trabajo general y trasladar ese conocimiento adquirido a una herramienta informática, pero sin tener en cuenta que determinadas decisiones de diseño pueden influir positiva o negativamente de cara a futuras adaptaciones de la herramienta.

  Para justificar el objetivo principal del TFM, que no es otro que añadir soporte a la edición colaborativa de *RCMs*, surge la necesidad de llevar a cabo una labor de reingeniería que permita dotar al proyecto de una estructura interna definida, flexible, mantenible y escalable, que facilite la utilización de modelos de datos que permitan la sincronización de datos en el servidor. También habrá que tener en cuenta el alcance de posibles mejoras funcionales e integraciones futuras de cualquier otro tipo.

  Las labores de reingeniería se centrarán en la continuidad del patrón de arquitectura propuesto *MVC*, incidiendo en una mejora de la separación entre capas aprovechando el *framework* de trabajo *Express.js*.

  Hay que tener en cuenta que, tras las labores de reestructuración, se deben mantener o en su defecto ampliar los requisitos funcionales con los que cuenta la herramienta en su versión inicial.

- **Redefinir el modelo de base de datos y definir nuevos modelos de datos**

  A nivel de base de datos la estructura de datos propuesta inicialmente en *MongoDB* se basa en la definición de una única colección de usuarios. Esta estructura de datos almacena tanto las credenciales de acceso del usuario a la aplicación como su listado de *RCMs* asociados.

A pesar de que la arquitectura inicial de *RCMTool* pretende acercarse a un patrón *MVC*, internamente no está definido un modelo de datos como tal. Existe un modelo que contiene los mecanismos para acceder a la información y realizar una serie de operaciones *CRUD* sobre la base de datos, pero trabaja directamente mediante consultas genéricas sobre *MongoDB*, en ningún momento define un esquema o modelo de datos concreto.

La simplicidad de la base de datos a la hora de gestionar la información está justificada porque, aun siendo bastante limitada, es lo suficientemente efectiva como para ser capaz de soportar las funcionalidades propuestas inicialmente.

Haciendo una exhausta valoración de las características que se pretenden pulir y teniendo en cuenta las nuevas funcionalidades a desarrollar, se antoja necesario definir un modelo de base de datos mucho más flexible y escalable con vistas a ampliaciones futuras.

Se propone la definición de un nuevo modelo de datos mediante el uso de esquemas en la capa de persistencia de la aplicación. De esta forma, se fijan una serie de restricciones y limitaciones que se van a ver reflejadas en la estructura de la base de datos *MongoDB* y van a facilitar las operaciones de acceso a datos.

- **Mejorar la usabilidad y facilitar la interacción con el usuario**

  En líneas generales, la interfaz gráfica de una aplicación de carácter académico no debería considerarse el factor más importante a tener en cuenta a la hora de valorar un desarrollo de software. Lo normal es que primen otros factores cuantitativos, como la funcionalidad o la disponibilidad, que inciden más directamente en aspectos medibles como la productividad. Sin embargo, cuando la usabilidad y la experiencia del usuario se ven afectadas por problemas derivados de una deficiente presentación de la información, es necesario plantearse un rediseño de la capa de presentación.

  RCMTool cuenta con una serie de vistas sencillas y estéticamente simples, pero hay que hacer especial hincapié en los elementos gráficos. El editor de

RCMs es una utilidad que aporta muchas posibilidades al usuario y el principal motor de la aplicación, por lo que es necesario que se adapte a sus necesidades. Uno de los puntos débiles del editor y del resto de vistas en general es su *look-and-feel*. Los elementos visuales no se proyectan correctamente y, por lo general, no se adaptan al tamaño del navegador.

Se valora positivamente la renovación del apartado gráfico. Las vistas son bastante limitadas y los estilos están poco cuidados. Se propone un rediseño de la capa de presentación que aporte frescura y mejore la experiencia del usuario.

- **Mejorar el sistema de manejo de errores**

  Uno de los puntos clave para mejorar la experiencia del usuario es mantenerle informado en todo momento del estado de la aplicación.

  Hasta ahora, siempre que se produce cualquier tipo de incidencia, independientemente de su origen, no se facilita ningún tipo de información al respecto. La aplicación debe ser capaz, en primer lugar, de informar al usuario del error que ha ocurrido y, en segundo lugar, de facilitar mecanismos que permitan subsanar el problema.

  Dividimos el origen de los errores en dos posibles motivos: errores internos de la aplicación, como puede ser una caída de la conexión con la base de datos, y errores ocasionados por limitaciones del sistema, como por ejemplo el alta de dos usuarios con un mismo nombre de usuario. En cualquier caso, siempre que produzca una variación del comportamiento habitual de la herramienta es necesario informar al usuario de los problemas que han originado esa actuación.

  La gestión de errores no debe limitarse exclusivamente en informar de este tipo de incidencias al cliente. También debe quedar internamente reflejada en el lado del servidor, registrando cualquier tipo de actividad en el log correspondiente de la aplicación. De esta forma, se facilitan las labores de soporte, ya que es posible llevar a cabo la trazabilidad de los movimientos realizados por un usuario en un determinado momento.

- **Añadir funcionalidades avanzadas**

  La herramienta cuenta con una serie de funcionalidades estándar que permiten al usuario realizar determinadas tareas básicas. Sin embargo, cuenta con algunas limitaciones y deficiencias detectadas que es necesario subsanar.

  Uno de los propósitos del TFM es el de desplegar la herramienta en un entorno de producción accesible a los usuarios finales. Para permitir el acceso a la aplicación, el usuario deberá autenticarse mediante unas credenciales de acceso. Actualmente, la aplicación no cuenta con un proceso de registro de usuarios. Se plantea necesario reformular la página de acceso a la aplicación para ofrecer la posibilidad de registro a nuevos usuarios.

  En cuanto al almacenamiento de información de claves de acceso y otros datos sensibles, se deben utilizar mecanismos de seguridad que permitan cifrar dicha información en la base de datos. A pesar de que se trata de una aplicación de carácter académico cuyo uso va a ser restringido, es necesario aportar instrumentos que permitan la recuperación de contraseñas sin poner en riesgo la identidad del usuario.

  Otra de las características que pueden suponer una mejora de la usabilidad de la herramienta es la validación de datos en los formularios. Se puede orientar con dos objetivos distintos: por un lado, informar a los usuarios de los campos que contienen información no válida, y por el otro, permite limitar el envío y procesamiento de formularios al servidor siempre y cuando la información introducida no sea correcta en su totalidad.

- **Actualizar los recursos utilizados en la medida de lo posible**

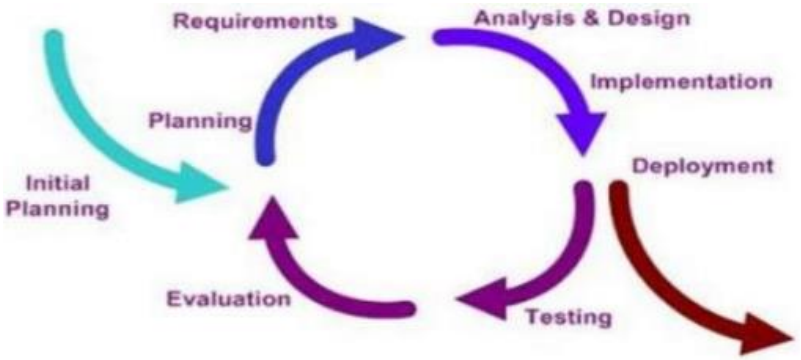
  RCMTool utiliza una gran variedad de tecnologías, además de un amplio número de librerías y módulos externos, tanto en el lado del cliente como en el lado del servidor. Desde su implantación han transcurrido algo más de dos años. Informáticamente hablando, a lo largo de este tiempo no sólo es probable que se hayan desarrollado nuevas herramientas de trabajo que simplifiquen en gran medida muchas de las operaciones que en su día se

implementaron, sino que las propias tecnologías han evolucionado en cuanto a funcionalidad, facilidad de uso y corrección de errores detectados, entre otros.

Es necesario realizar un estudio del alcance que tiene el uso de cada una de las dependencias externas del proyecto y actualizar aquellos recursos que se estimen oportunos.

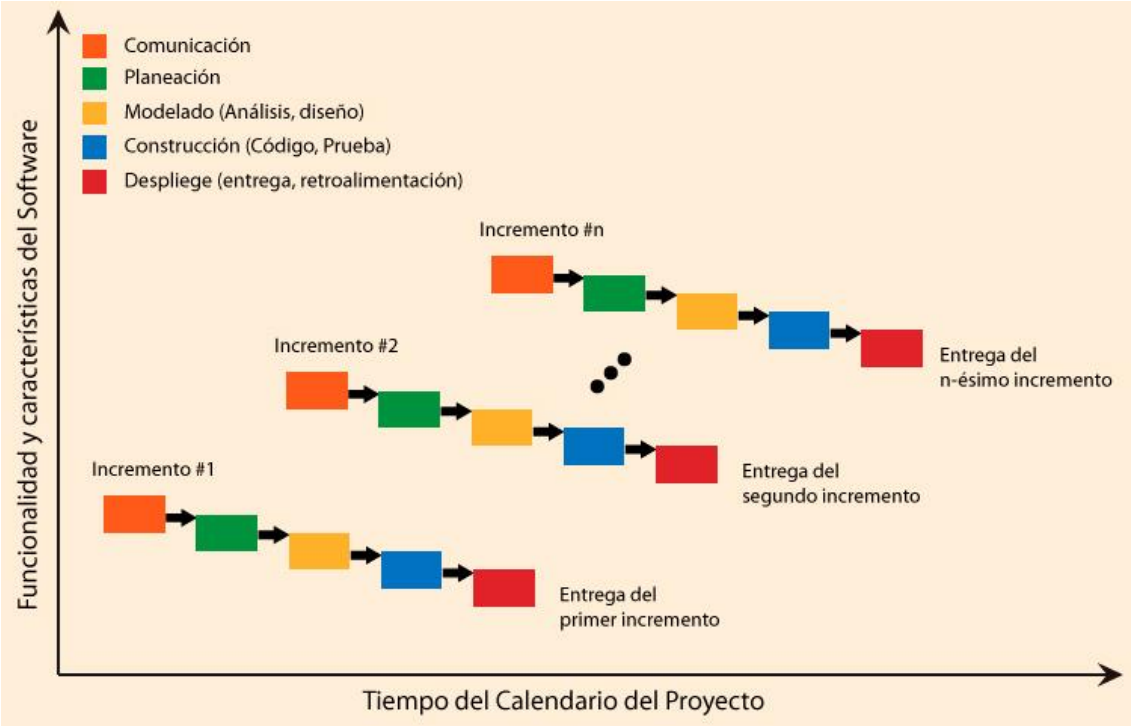
Plan de trabajo

El marco de trabajo seguido para estructurar, planificar y controlar el proceso de desarrollo se ha basado en un modelo de desarrollo iterativo e incremental.



[Figura 3] Modelo de desarrollo iterativo

Este modelo de desarrollo [Figura 3] [Figura 4] de software se enfoca en la construcción de secciones reducidas de software que van ganando complejidad, facilitando así la detección de problemas en fases tempranas.



[Figura 4] Gráfica de desarrollo incremental

Desarrollo del trabajo

Una vez expuestos los motivos objeto del proyecto y la metodología de desarrollo a seguir, es necesario realizar la planificación de los puntos de actuación en el tiempo.

Considerando los desencadenantes que puede conllevar la realización de cada una de las tareas en relación a cómo afectan al resto de tareas del proyecto, se establece el siguiente orden a la hora de proceder:

1. Reestructuración interna
2. Actualización de recursos
3. Funcionalidades avanzadas
4. Mejora del manejo de errores
5. Rediseño gráfico
6. Revisión de la gestión de referencias
7. Definición del modelo de datos
8. Integración con gestores de referencias externos
9. Edición colaborativa

Conviene hacer hincapié en que el orden de actuación definido no es consecuente con el nivel de prioridad establecido para cada una de las tareas/objetivos del proyecto. El objetivo prioritario es conseguir una herramienta que permita el trabajo colaborativo con *RCMs*. Sin embargo, el proceso para conseguir esta funcionalidad es abordar inicialmente una serie de tareas previas, orientadas en adecuar la estructura interna de la aplicación hacia un entorno de trabajo con la lógica de negocio alojada en el servidor, que permitan adaptar la aplicación para conseguir el fin último.

1. Reestructuración interna

La estructura inicial de la herramienta intenta aproximarse a un patrón de arquitectura Modelo-Vista-Controlador, o *MVC*. A pesar de ello, la integración entre capas está forzada por el diseño impuesto en su desarrollo, sin contar con

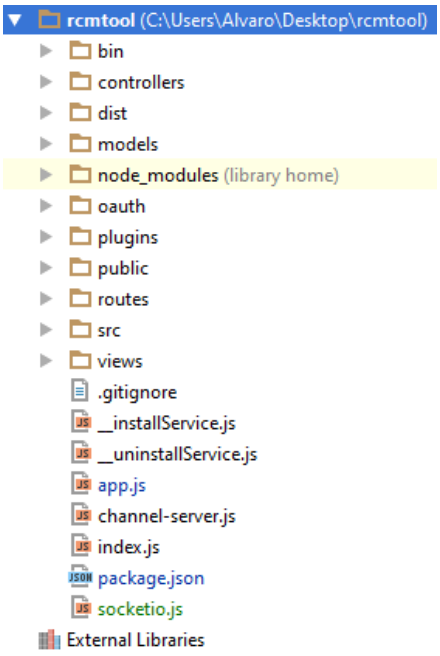
ningún tipo de soporte para tal objetivo por parte del *framework* de trabajo *Express.js*, en su versión *3.19.2*.

Como todo desarrollo software, independientemente de la metodología del ciclo de trabajo utilizada, la funcionalidad de un proyecto no está definida en su totalidad desde el principio. El propio modelo de desarrollo ofrece una serie de mecanismos que permiten aportar mejoras y nuevas funcionalidades a lo largo del ciclo de vida de la aplicación. Las infraestructuras de trabajo, en este caso *Express.js*, no escapan de esta definición y son capaces de evolucionar en el tiempo para adaptarse a nuevas metodologías de trabajo.

A partir de la versión *4.x*, *Express.js* incorpora una serie de cambios en su generador de aplicaciones que permite crear rápidamente un esqueleto de aplicación orientada a la separación entre capas. Otros cambios importantes de esta nueva versión están relacionados con el sistema principal y de *middleware* que utiliza *Express.js*, y cambios en el sistema de direccionamiento, entre otros.

Se establece como prioridad la actualización de *RCMTool* a la última versión estable de *Express.js*, la *4.13.4* al inicio de esta fase. Sin embargo, hay que tener en cuenta que, tal y como se indica en la página web oficial de la infraestructura web [13], la migración a *Express 4* supone un cambio que rompe con el código existente de *Express 3*. Esto implica que una aplicación *Express 3* existente no funcionará si actualiza la versión de *Express* en sus dependencias. En consecuencia, es imprescindible partir de un nuevo proyecto para la reestructuración de la herramienta.

La estructura interna del nuevo proyecto queda definida de la siguiente forma, dando bastante importancia a la separación entre capas [Figura 5].



[Figura 5] Estructura interna del proyecto

Gestión de controladores

Para organizar las labores de programación, se ha definido un modelo base que va a simular la herencia entre clases. La utilización de mecanismos de herencia permite favorecer el mantenimiento y la extensión de la aplicación.

El fichero *BaseController.js* [Figura 6] simula una clase genérica abstracta de la que van a extender el resto de controladores. Utiliza la librería *UnderScore.js* [14] para permitir la herencia. La clase tiene tres propiedades:

- *name*: identifica el nombre del controlador. Cada uno de los controladores que hereda de esta clase redefinirá su propio nombre.
- *extend*: permite la herencia. Cada controlador hijo utilizará esta propiedad para redefinir los valores de sus propiedades *name* y *run*.
- *run*: permite redefinir la funcionalidad concreta de cada controlador.

BaseController.js

```
var _ = require('underscore');

module.exports = {
    name: 'Base Controller',
    extend: function(child) {
        return _.extend({}, this, child);
    },
    run: function(req, res, next) {
    }
};
```

[Figura 6] Contenido *BaseController.js*

El resto de controladores extenderán de este modelo y reimplementarán su funcionalidad [Figura 7]:

Index.js

```
var path = require('path');

var BaseController = require(path.join(__dirname, '..', 'BaseController'));
var BaseView = require(path.join(__dirname, '..', '..', 'views', 'BaseView'));

module.exports = BaseController.extend({
    name: 'Index',
    run: function(req, res, next) {

        var textmessage = req.session.textmessage;
        delete req.session.textmessage;

        var view = new BaseView(res, 'index');
        view.render({
            title: 'RCMTool - Sign in',
            partials: {
                header: 'partials/header',
                footer: 'partials/footer',
                modalmessage: 'modals/message'
            },
            titlemessage: 'Info Message',
            textmessage: textmessage
        });

    }
});
```

[Figura 7] Controlador heredado

Comprobamos que el controlador redefine sus propiedades *name* y *run*, mientras que utiliza la propiedad *extend* para indicar que la clase extiende su funcionalidad del controlador base [Figura 7].

Gestión de vistas

El fichero *BaseView.js* [Figura 8] simula una clase genérica que van a utilizar los controladores para redireccionar las peticiones a las vistas. Utiliza el mecanismo de prototipado de *Node.js* para añadir funcionalidad a la clase.

BaseView.js

```
function BaseView(response, template) {
    this.response = response;
    this.template = template;
}

BaseView.prototype.render = function(data) {
    if (this.response && this.template) {
        this.response.render(this.template, data);
    }
};

module.exports = BaseView;
```

[Figura 8] Contenido *BaseView.js*

En la imagen anterior, podemos ver como el controlador utiliza la clase *BaseView.js* para gestionar la visualización de la página [Figura 8].

Gestión de modelos

Cada modelo [Figura 9] va a definir un conjunto de propiedades o funciones que van a trabajar directamente sobre sus esquemas.

ReferenceModel.js

```
var path = require('path');

var Reference = require(path.join(__dirname, 'Reference'));

module.exports = {
    insert: function(callback, data) {

        Reference.findOneAndUpdate({ key: data.key, source: data.source }, data, { upsert: true, new: true }, callback);

    },
    getReference: function(callback, _id) {

        Reference.findById(_id, 'source title type authors year abstract', callback);

    }
};
```

[Figura 9] Modelo de un esquema

Más adelante veremos con mayor nivel de detalle cómo se gestiona la dependencia entre los modelos, sus esquemas y la estructura de la base de datos.

2. Actualización de recursos

Podemos dividir las tecnologías del proyecto en función del área en el que tienen influencia: en el lado del servidor o en el lado del cliente.

Servidor

El entorno de ejecución *Node.js* utiliza el gestor de paquetes *npm* para la instalación de módulos y dependencias. El concepto de módulo es un pequeño programa que contiene funciones, objetos y variables que serán exportadas para ser utilizadas en otros programas (lo que se conoce comúnmente en otros lenguajes de programación como librerías). Los desarrolladores pueden crear, compartir y reutilizar módulos de sus aplicaciones.

Npm utiliza un fichero *JSON*, *package.json*, para gestionar el versionado de dependencias de un proyecto. Es el propio gestor el encargado de descargar del repositorio de *Node.js* las últimas versiones disponibles de cada módulo en el directorio *node_modules* del proyecto.

Cliente

RCMTool cuenta con una serie de librerías externas que permiten ampliar las funcionalidades básicas de *JavaScript*. Al contrario de lo que ocurre en el lado del servidor, la gestión del versionado de librerías la debe realizar el desarrollador.

Se han actualizado las versiones en la medida que ha sido posible. Sin embargo, han surgido una serie de problemas a la hora de actualizar algunas librerías. Recordamos que la librería *Rappid* es de código cerrado y fue cedida en su día para la realización del proyecto por considerarse este con fines académicos. Es por ello que las fuentes de *Rappid* no han sido actualizadas. Esta librería tiene una serie de dependencias, entre las que se encuentra *jQuery*. En consecuencia, no ha sido posible actualizar *jQuery* más allá de la versión *1.9.1* por problemas de colisiones con la versión que disponemos de *Rappid*.

3. Funcionalidades avanzadas

Acceso a la aplicación

Comenzando por la página principal de acceso a *RCMTool*, se ha modificado la vista para contener un formulario dinámico que permita, tanto el acceso a la aplicación, como el registro de nuevos usuarios.

Cifrado de datos vulnerables

Otro punto en el que se ha hecho hincapié es en el cifrado de datos sensibles en la base de datos. En concreto, se utilizan distintos mecanismos para almacenar cifradas las claves de acceso de los usuarios a la aplicación y sus *tokens* de refresco proporcionados por *Mendeley*. Como veremos más adelante, *Mendeley* ofrece un mecanismo de refresco de claves de acceso en el que no

es necesario que intervenga el usuario. Se trata, por tanto, de un dato susceptible de ser almacenado de forma segura.

Para entender el cifrado que se aplica sobre la contraseña de acceso al sistema [Figura 10], es necesario entender que *MongoDB* identifica cada uno de los documentos de su base de datos con un *_id* interno único. En caso de que no se especifique uno, *MongoDB* genera uno automáticamente.

El cifrado de la clave de acceso [Figura 10] durante el registro se lleva a cabo de la siguiente forma:

a. Se realiza la inserción del documento que hace referencia al usuario. *MongoDB* asigna automáticamente un identificador interno (*_id*) al documento, almacenando la contraseña inicialmente en claro.
b. Se calcula un *hash* numérico utilizando como semillas el *_id* asignado al usuario, el nombre del usuario y la contraseña utilizada durante el proceso de registro.
c. Se almacena el *hash* calculado en el documento que hace referencia al usuario en *MongoDB*.

```
encodePassword: function(callback, data) {

    var calculateHash = hash([ data._id.toString(), data.username, data.password ], { algorithm: 'sha1' });
    User.findByIdAndUpdate(data._id, { $set: { password: calculateHash }}, callback);

},
```

[Figura 10] Cifrado de la clave de acceso

El proceso de autenticación cuando un usuario intenta acceder a la aplicación es el siguiente:

a. El usuario introduce su nombre de usuario y su clave de acceso en claro.
b. El controlador correspondiente recupera el documento cuyo nombre de usuario corresponde al proporcionado. Hay que tener en cuenta que el nombre de usuario es único, por lo que únicamente puede tener asignado un identificador interno (*_id*).
c. Se realiza el cálculo del *hash* resultante utilizando como semillas el *_id* del documento, el nombre de usuario y la contraseña introducida.

d. Se comparan ambas cadenas. Si el *hash* calculado se corresponde con el que está almacenado en la base de datos, el proceso de verificación es correcto. De lo contrario, la contraseña es incorrecta.

Para realizar el cálculo del *hash* se utiliza el módulo *object-hash* [15] que proporciona *Node.js*.

El proceso de encriptación del *token* de refresco de *Mendeley* [Figura 11] es algo más complejo. En este caso, nos servimos del módulo *crypto* [16] de *Node.js*.

El procedimiento de encriptación [Figura 11] es de la siguiente forma:

a. Se calcula el *hash* tomando como semillas el *_id* del usuario y su nombre de usuario, utilizando una función resumen *SHA1*.
b. Se instancia un elemento de cifrado utilizando un algoritmo *AES-256* basado en un modo *CTR* y, como clave, el *hash* generado.
c. Se actualiza la instancia de cifrado utilizando el *token* de refresco de *Mendeley*.
d. La petición devuelve un objeto encriptado cuya salida se ofrece en formato hexadecimal. El valor obtenido es el valor original encriptado, que es el que se almacena en la base de datos.

```
updateMendeleyRefreshToken: function(callback, data) {

    var calculateHash = hash([ data._id.toString(), data.username ], { algorithm: 'sha1' });
    var cipher = crypto.createCipher('aes-256-ctr', calculateHash);
    var crypted = cipher.update(data.refreshToken, 'utf8', 'hex');
    crypted += cipher.final('hex');
    User.findByIdAndUpdate(data._id, { $set: { mendeleyRefreshToken: crypted } }, { new: true }, callback);

},
```

[Figura 11] Encriptación del *token* de refresco de *Mendeley*

El procedimiento de desencriptación es idéntico, pero instanciado un elemento de descifrado y utilizando los mismos algoritmos y funciones resumen.

Validación de formularios

Node.js se trata de una tecnología que responde de manera muy eficiente al uso de peticiones asíncronas, especialmente cuando se trata de peticiones ligeras que apenas suponen tiempo de procesamiento en el servidor.

La librería *jQuery Validation Plug-in* [17], además de una serie de validaciones propias en el lado del cliente, permite el uso de peticiones *AJAX* para realizar validaciones en el servidor, sin necesidad de procesar formularios de datos completos.

Se pretende aprovechar el rendimiento que ofrece *jQuery Validation* para realizar validaciones asíncronas previo envío de datos al servidor, controlando el procesamiento de formularios síncrono únicamente en aquellos casos en los que la información introducida sea válida en su totalidad.

De esta forma, limitamos el número de llamadas síncronas al servidor, minimizamos el tiempo de respuesta y mejoramos la usabilidad del usuario.

4. Mejora del manejo de errores

Una aplicación *Express.js* es fundamentalmente una serie de llamadas a funciones de *middleware*. Las funciones de middleware sin funciones que tienen acceso al objeto solicitud *req*, al objeto respuesta *res* y a la siguiente función de middleware *next*, en el ciclo de solicitud/respuesta de la aplicación.

Express.js permite gestionar el manejo de errores mediante funciones *middleware* [Figura 12]. A excepción del resto, este tipo de funciones tienen cuatro argumentos: (*err, req, res, next*).

```
// Manejador de errores del entorno de desarrollo
// Middleware que se ejecuta siempre que se propaga un error desde cualquiera de los controladores
if (app.get('env') === 'development') {
    app.use(function(err, req, res, next) {
        console.error(err.errmsg);
        req.session.error = err.errmsg;
        res.redirect('/error');
    });
}
```

[Figura 12] Manejo de errores en *Express.js*

Aprovechando esta funcionalidad, tanto los enrutadores *routers* como los controladores de la aplicación pueden gestionar cualquier error que se produzca en el lado servidor. Normalmente, este tipo de errores suelen estar relacionados con errores internos.

Siempre que se produzca un error de este tipo, la aplicación dejará constancia en el log de errores de la aplicación del evento que se ha producido y redirigirá al usuario a una página en la que le informará de dicho error.

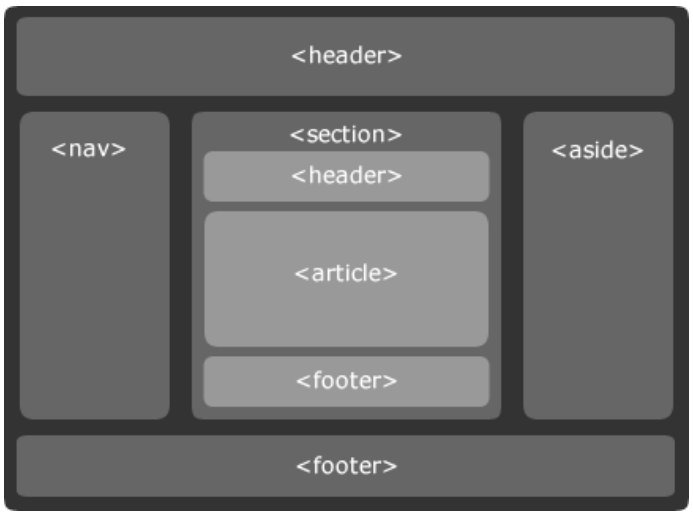
El procedimiento fijado para aquellos casos en los que se produzca un error en el lado del cliente se gestiona de forma distinta. Por lo general, los errores producidos serán ocasionados por limitaciones impuestas sobre el uso de la herramienta. En esos casos, se deberá informar al usuario del error o restricción oportuna a través de un mensaje en la zona inferior de la página (si estamos en el editor de *RCMs*) o mediante un diálogo modal emergente (si nos encontramos en cualquier otra vista).

En cualquier caso, un error producido del lado del cliente no dejará traza en el log de errores de la aplicación a menos que sea como consecuencia de una petición asíncrona generada desde el cliente.

5. Rediseño gráfico

Para llevar a cabo el *restyling* se ha requerido previamente de un estudio de las diferentes vistas de las que iba a disponer la aplicación. Partiendo de esa base, se han maquetado cada una de las vistas individualmente para que todas ellas siguieran un mismo estilo base.

La estructura base de las páginas parte de la siguiente distribución:



[Figura 13] Estructura web base

Partiendo de la distribución anterior, se han definido tres tipologías de vista distintas:

- Acceso/registro
  Es la vista inicial que se muestra al usuario cuando accede por primera vez a la aplicación. Consta de tres elementos, combinando el cuerpo de la página y sus menús laterales en un único elemento central.

- Menú de usuario
  Comprende los distintos submenús de trabajo del usuario, una vez autenticado en la aplicación. Queda dividido en cuatro elementos, suprimiendo el elemento *aside*.

- Editor
  Permite la edición gráfica de *RCMs*. Se encuadra en la estructura de normalizada cinco elementos.

La paleta de colores elegida se ha orientado teniendo en cuenta que se trata de una herramienta de ámbito académico. Se han buscado colores y tonalidades neutras que permitiesen resaltar algunos de los menús, pero consiguiendo una combinación visualmente atractiva.

Para mejorar la usabilidad del usuario se ha utilizado la biblioteca de componentes *jQuery UI* del *framework jQuery*, que ofrece un conjunto de *plug-ins widgets* y efectos visuales disponibles para añadir comportamientos complejos a los elementos de la página.

6. Revisión de la gestión de referencias

Surge el concepto de referencia como un modelo de datos [Figura 14] con entidad propia. Como tal, aparecerá reflejada como una colección en la base de datos.

Reference.js

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var referenceSchema = new Schema({
    key: {
        type: String,
        required: true
    },
    source: String,
    title: String,
    type: String,
    authors: [{
        _id: false,
        first_name: String,
        last_name: String
    }],
    year: String,
    abstract: String
});

referenceSchema.index({ key: 1, source: 1}, { unique: true, sparse: true });

module.exports = mongoose.model('Reference', referenceSchema);
```

[Figura 14] Esquema de *Reference*

Cada referencia bibliográfica aparecerá una única vez instanciada en la base de datos con un identificador interno único.

Como consecuencia, cualquier elemento de un *RCM* que se relacione con una determinada *Reference*, como un *Path* o el propio *RCM*, deberán utilizar su *_id* para referirse a ella.

Por otra parte, surge la necesidad de relacionar cada referencia de alguna manera con el usuario que la utiliza. La idea es que el usuario cuente con un conjunto de referencias propias y asociadas a él, que pueda seleccionar para incorporarlas en un determinado momento a un *RCM*.

Fruto de esta necesidad, surge el concepto de carpeta como contenedor de referencias. Una carpeta también hace referencia a un modelo de datos con entidad propia.

Conceptualmente, la forma de entender la asignación entre carpetas y referencias es la misma que utiliza *Mendeley* para la gestión interna de referencias.

Folder.js

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var folderSchema = new Schema({
    key: String,
    source: String,
    name: {
        type: String,
        required: true
    },
    referenceList: [{
        type: Schema.Types.ObjectId,
        ref: 'Reference'
    }]
}, {
    timestamps: true
});

folderSchema.index({ key: 1, source: 1}, { unique: true, sparse: true });

module.exports = mongoose.model('Folder', folderSchema);
```

[Figura 15] Esquema de *Folder*

Cada carpeta aparecerá también una única vez instanciada en la base de datos [Figura 15] con un identificador interno único.

La forma de identificar las referencias de cada usuario será a través de sus carpetas. Es decir, un usuario tiene asociada una lista de carpetas. Cada una de estas, a su vez, contiene un conjunto de referencias. Por tanto, las referencias que pertenezcan a las carpetas asociadas al usuario, serán las referencias con las que pueda trabajar el usuario.

Cada vez que un usuario proceda a dar de alta un *RCM* en la aplicación, la aplicación permitirá al usuario seleccionar el contenido de las carpetas que quiere asociar al *RCM*. Sin embargo, los *RCM* trabajarán directamente sobre el listado de referencias. Las carpetas únicamente tienen sentido como unidad para agrupar las referencias de cara a la organización del usuario.

RCMTool ofrece la posibilidad de añadir tantas referencias a la aplicación, como sea necesario. La principal diferencia es que, en vez de asociarlas directamente al *RCM*, ahora las asocia a una carpeta del usuario. Puede darse el caso de que un usuario tenga referencias cargadas en la aplicación, pero no las use en ninguno de sus *RCMs*.

Los mecanismos para añadir referencias a la aplicación son los siguientes:

- Referencias en formato *BibTeX*.
  La aplicación permite incorporar directamente el texto formateado a través de diálogos modales emergentes.

- Importación de ficheros de referencias externos.
  La aplicación permite la carga de ficheros en texto plano que contengan un listado de referencias, también en formato *BibTeX*.

- Gestor de referencias de *Mendeley*.
  RCMTool se integra con *Mendeley* para listar el conjunto de carpetas que tiene asociadas el usuario en el gestor de referencias. Sobre ese listado, permite seleccionar aquellas carpetas de las que se pretende descargar su contenido en la aplicación.

7. Definición del modelo de datos

La estructura de datos inicialmente propuesta se basa en la definición de una única colección de usuarios del sistema. Cada usuario tiene asociadas unas credenciales de acceso y el listado de *RCMs* que ha creado. Se trata de una base de datos algo simplista a la hora de gestionar la información, pero que es lo suficientemente efectiva para soportar las funcionalidades de esta primera versión. Para el acceso a la capa de persistencia, *Node.js* utiliza las librerías nativas del propio sistema gestor de base de datos *MongoDB*.

El desarrollo de nuevas funcionalidades y, por tanto, de la complejidad interna del proyecto y de sus modelos, acompañado de un proceso de reestructuración interna y de un cambio en la forma en que *RCMTool* gestiona las referencias, ha generado la necesidad de rediseñar el modelo de datos de la aplicación.

Para dar soporte al modelado de objetos en entornos asíncronos se propone el uso del módulo *Mongoose.js*. Se trata de un módulo de *Node.js* que facilita la interacción de operaciones *CRUD* con bases de datos *NoSQL* como *MongoDB*.

Mongoose.js permite modelar la estructura de una base de datos mediante la definición de esquemas. Cada esquema se asigna a una colección *MongoDB* y define la estructura de los documentos y sus propiedades dentro de esa colección [Figura 14] [Figura 15].

Desde *Node.js* se debe generar el modelo correspondiente encargado de gestionar las operaciones *CRUD* con la base de datos [Figura 16], utilizando los esquemas definidos en *Mongoose.js*.

ReferenceModel.js

```
var path = require('path');

var Reference = require(path.join(__dirname, 'Reference'));

module.exports = {
    insert: function(callback, data) {

        Reference.findOneAndUpdate({ key: data.key, source: data.source }, data, { upsert: true, new: true }, callback);

    },
    getReference: function(callback, _id) {

        Reference.findById(_id, 'source title type authors year abstract', callback);

    }
};
```

[Figura 16] Modelo de *Reference*

Se propone la definición de cinco esquemas, cada uno con su respectivo modelo en *Node.js* para gestionar sus operaciones de acceso a base de datos:

- *User*

_id	Identificador interno
firstname	Nombre
lastname	Apellido/s
email	Correo electrónico
avatar	Imagen binaria del avatar
username	Nombre de usuario
password	Contraseña
rootFolder	Identificador de carpeta raíz La carpeta raíz es una carpeta asignada por defecto al usuario y que contiene el listado de identificadores del conjunto de referencias asociadas al usuario. Se crea automáticamente cuando se da de alta un nuevo usuario.
folderList	Lista de identificadores de carpeta asociadas
createdAt	Fecha de creación
mendeleyRefreshToken	Token de refresco para la sincronización con *Mendeley*

- *Folder*

_id	Identificador interno
key	Clave interna de la carpeta que la identifique en un gestor de referencias externo (como, por ejemplo, *Mendeley)*. En caso de que se trate de una carpeta creada desde el sistema, la clave será vacía.
source	Fuente de la carpeta. Permite identificar la fuente de alta de la carpeta, distinguiendo si se ha cargado desde *Mendeley* o se ha creado nueva desde el propio sistema.
name	Nombre
referenceList	Lista de identificadores de referencias asociadas
createdAt	Fecha de creación
updatedAt	Fecha de última actualización

- *Reference*

_id	Identificador interno
key	Clave interna de la referencia que la identifique en un gestor de referencias externo (como, por ejemplo, *Mendeley*). En caso de que se trate de una referencia importada desde un fichero, la clave será vacía.
source	Fuente de la referencia. Permite identificar la fuente de alta de la referencia, distinguiendo si se ha cargado desde *Mendeley* o desde otro origen de datos (por ejemplo, un fichero de carga).
title	Título
type	Tipo
authors	Lista de autores
year	Año
abstract	Resumen

- *Path*

_id	Identificador interno
index	Índice de la referencia en el *RCM* (1, 2, 3, etc.)
suffix	Sufijo que identifica individualmente a cada *Path* del resto de *Paths* que están asociados a una misma Reference. El orden de asignación viene determinado por un carácter consecutivo (“a”, “b”, “c”, etc.)
text	Texto de la definición
reference	Identificador de la referencia asociada
firstConcept	Identificador del concepto principal en el grafo
syntagmList	Lista de los sintagmas que contiene el *Path*.

	Cada sintagma está compuesto por pares de identificadores: id del *LinkingPhrase* – id del *Concept* sucesivo.

- *RCM*

_id	Identificador interno
name	Nombre
description	Descripción
creator	Identificador del usuario creador
userList	Lista de identificadores de usuarios con acceso
referenceList	Lista de identificadores de referencias asociadas
pathList	Lista de identificadores de *Paths* asociados
svg	Objeto *SVG* del grafo en formato *JSON*
conceptList	Lista de identificadores de *Concepts* del grafo
linkingPhraseList	Lista de identificadores de *LinkinsPhrases* del grafo
createdAt	Fecha de creación
updatedAt	Fecha de última actualización

8. Integración con gestores de referencias externos

Mendeley ofrece un *API* de conexión propia que facilita el acceso a su información. Utiliza el protocolo de autorización *OAuth2* [12] que permite a aplicaciones de terceros acceder a contenidos propiedad de un usuario alojados en otras aplicaciones de confianza (servidor de recursos), sin que las aplicaciones de terceros tengan que manejar ni conocer las credenciales del usuario.

El sentido de utilizar *OAuth2* es que soluciona el problema de la confianza entre un usuario y aplicaciones externas. A su vez, permite a un proveedor de servicios/*API* facilitar a aplicaciones de terceros que amplíen sus servicios haciendo uso de los datos de sus usuarios de manera segura y dejando al usuario la decisión de cuándo y a quién, revocar o facilitar el acceso a sus datos, creando así un ecosistema de aplicaciones alrededor del proveedor de servicios/*API*.

Mendeley ofrece varios sistemas de autorización y acceso a datos. La forma más común de utilizar sus servicios/*API* [11] es utilizar el *SDK* propio del lenguaje de programación que se utiliza desde el proyecto cliente. Para ello ofrece su *SDK*

en una serie de lenguajes de programación bastante extendidos: *Ruby*, *JavaScript*, *C++*, *Python*, *Java*, etc.

Hay que tener en cuenta que la versión inicial de *RCMTool*, concluida a finales de 2014, llegó a utilizar el *SDK* de *JavaScript* para interactuar con *Mendeley*. Durante ese período y hasta que dio comienzo mi TFM, en torno a principios del año 2016, el *SDK* sufrió una serie de cambios y modificaciones que dejaron totalmente inservible la integración existente con *Mendeley*.

En base a esta experiencia, se ha optado por implementar la integración con utilizando la *API* a través del protocolo *HTTP*. El motivo ha ido en consonancia al principal inconveniente de utilizar la *SDK*: ofrece una mayor integración, pero requiere de un mantenimiento constante. Hay que tener en cuenta que la *SDK* cambia de un día para otro sin aviso previo y hace necesario la revisión/modificación del procedimiento completo de integración. *RCMTool* es una herramienta que, de momento, no se prevé que cuente con soporte diario, por lo que se ha decidido optar por un entorno más estable y mantenible. El *API* ofrece mayor estabilidad puesto que es menos propenso al cambio, al menos en este caso concreto que nos concierne.

Mendeley utiliza un portal para desarrolladores propio que permite gestionar la obtención de *tokens* de acceso para autorizar las peticiones a su *API*.

El método de registro es simple. Una vez autenticado, cualquier usuario tiene acceso a un portal dentro de la aplicación que le permite autorizar sus propias aplicaciones de terceros para que tengan acceso al *API* de *Mendeley*. Basta con indicar el nombre de la aplicación, una breve descripción de su funcionamiento e indicar la *URL* de respuesta (de nuestra aplicación) que será la encargada de procesar la respuesta. Automáticamente, *Mendeley* genera un identificador de aplicación y una clave secreta que son los que utilizará la aplicación externa para autenticarse.

El proceso de autenticación es el siguiente:

a. Desde la aplicación externa, se invoca a la *URL* del servicio de autenticación *Oauth*.

b. Se muestra la interfaz de autenticación del usuario.
c. *Mendeley* captura los parámetros de autenticación de la interfaz de usuario y, en base a los datos recibidos, genera una respuesta.
d. Se produce una redirección hacia la *URL* de la aplicación externa que, recordamos, se ha definido desde el portal de aplicaciones de *Mendeley*.
e. La aplicación externa procesa la respuesta y extrae los *tokens* de acceso y refresco.

Mendeley utiliza dos tipos de *tokens*, uno de acceso a la aplicación y otro de refresco. El primero de ellos lo utiliza como *token* de autenticación cuando se genera una petición a su *API*, mientras que el segundo lo utiliza como *token* de refresco para renovar el *token* de acceso. Como curiosidad, comentar que ambos *tokens* permanecen invariables a lo largo del ciclo de vida de la aplicación y no varían a lo largo del tiempo.

RCMTool almacena ambos *tokens* en las *cookies* del cliente. Sin embargo, las cookies, al igual que las variables de sesión, caducan cada cierto tiempo. Por este motivo, siempre que caduca el *token* de acceso es necesario volver a autenticarse en *Mendeley*. Ahí es donde entra en juego el *token* de refresco que, al contrario que el *token* de acceso, se almacena como variable de sesión del usuario y se persiste en la base de datos *MongoDB* (ya hemos comentado previamente que no se almacena en claro, sino que se utiliza un método de encriptación *SHA1*).

En el momento en que *RCMTool* detecta que el intercambio de datos con la interfaz de autenticación de *Mendeley* no es satisfactorio debido a que no dispone de un *token* de acceso, recurre a buscar el *token* de refresco asociado al usuario. Si el usuario dispone de uno, *RCMTool* lo utiliza para generar un *token* de acceso nuevo y permitir el intercambio de datos con la *API* de *Mendeley*.

A nivel de usuario la interacción desde la aplicación con *Mendeley* es simple, dado que el proceso de intercambio de datos es totalmente transparente. El usuario únicamente debe facilitar las credenciales de autenticación en *Mendeley* la primera vez que accede a la interfaz de acceso. A partir de ahí, aparecerá el listado de carpetas que tiene asociado el usuario en el gestor de referencias. El

usuario tiene la opción de seleccionar el conjunto de carpetas que desea importar en la aplicación y el programa es el encargado de descargar su contenido.

La estructura que utiliza *Mendeley* para gestionar su contenido es idéntica a la que utiliza *RCMTool* para la gestión de referencias. En ambas plataformas, el concepto de carpeta se utiliza como instrumento de agrupamiento de un conjunto de referencias.

9. Edición colaborativa

La funcionalidad que aporta mayor valor a la herramienta y la diferencia del resto es el editor de *RCMs*. La potenciación del factor colaborativo es el principal punto de actuación y sobre el que más recursos se han destinado durante su desarrollo. Es por ello que el principal ámbito de investigación del proyecto se ha orientado hacia la necesidad de establecer un entorno de trabajo en equipo.

Un sistema de software colaborativo es aquel que permite a un grupo de usuarios trabajar en entornos comunes simultáneamente, compartiendo información entre sí de manera ordenada y controlada. *RCMTool* se plantea como una herramienta de trabajo para la creación y edición de *RCMs*, en la que un usuario puede compartir su trabajo con el resto de la comunidad.

Ante ese nuevo paradigma, surge la posibilidad de que un mismo *RCM* pueda ser accesible por varios usuarios de la aplicación de forma paralela, permitiendo la edición concurrente. Este cambio en la manera de trabajar implica que los cambios realizados por un usuario en el editor deben ser visualizados por el resto de usuarios que estén trabajando sobre ese mismo *RCM* en tiempo real.

El propósito general para trabajar con más de una instancia de un mismo elemento desde varias máquinas clientes pasa por que todas ellas estén sincronizadas con una instancia global en el lado del servidor. De esta forma, un cambio sobre el elemento en uno de los clientes debe modificar el estado de ese elemento en el servidor. El resto de clientes pueden obtener en cualquier momento el estado actualizado del elemento mediante una operación de refresco sobre elemento global del servidor. Aplicando este concepto en *RCMTool*, es necesario que los *RCM* se gestionen del lado del servidor.

Como ya se ha comentado en varias ocasiones a lo largo del documento, la tecnología *Node.js* se comporta de manera muy eficiente al uso de peticiones asíncronas. Aprovechando esta característica y la idea expuesta de gestionar los *RCM* en el lado del servidor, el fin es aligerar el contenido de las vistas para que sea el servidor el encargado de gestionar la mayor parte de la lógica de negocio.

La idea es que las vistas se limiten a la realizar una serie de peticiones de datos asíncronas y, aprovechando la funcionalidad de los *callbacks* que utiliza el lenguaje *JavaScript*, muestren los datos actualizados al usuario. Los resultados son vistas mucho más ligeras que se va a mejorar la usabilidad del usuario y que van a reducir considerablemente los tiempos de carga de las páginas.

En consecuencia, también se van a mejorar los tiempos de respuesta del servidor. Uno de los puntos débiles de *RCMTool* son los tiempos de espera cuando las llamadas al servidor *Node.js* requieren de tareas de procesamiento complejas. *Node.js* ejecuta todas las peticiones de forma asíncrona a través de su único hilo de ejecución. Si el volumen de peticiones es alto, las tareas se van añadiendo a la cola. Cuanto mayor es el volumen de procesamiento de datos que requiere la aplicación en el servidor, mayor es el tiempo de respuesta al cliente. Siguiendo esta filosofía, es preferible peticiones asíncronas ligeras y que no afectan de forma directa en el tiempo de respuesta al cliente.

Desde un punto de vista más técnico, siempre que se produzca un cambio que altere cualquiera de los elementos del grafo o de sus listas asociadas, tendrá lugar la petición *AJAX* oportuna desde el lado del cliente para que los cambios se vean reflejados en el servidor. Automáticamente, deben desencadenarse las peticiones de refresco necesarias en el resto de clientes que estén consumiendo ese *RCM* en el momento actual para que vean reflejados los cambios en tiempo de ejecución.

Las consecuencias que puede suponer en la capa de negocio de *RCMTool* se traducen en un aumento considerable del número de controladores, que deberán gestionar un gran número de peticiones *AJAX*. Los enrutadores de la aplicación, serán los encargados de gestionar las redirecciones de todas esas peticiones para que se permitir el intercambio de datos entre los controladores y las vistas.

En consecuencia, será necesario adaptar los modelos de datos de la aplicación para que faciliten el acceso a datos.

El procedimiento utilizado por un usuario para habilitar la compartición de un *RCM* con cualquier otro usuario de la aplicación se fundamenta en el uso de identificadores. Cada usuario tiene asignado un identificador único propio, que corresponde al *_id* interno del documento al que hace referencia en la base de datos, que le reconoce del resto y es accesible desde el menú *Settings* de la herramienta. Desde la vista del editor de *RCMs*, el sistema permite la inserción de identificadores, valores hexadecimales de 24 caracteres de longitud, a través de un diálogo modal. La herramienta comprobará que el identificador introducido corresponde a un valor válido y permitirá el acceso al RCM por parte del usuario. Este método permite garantizar el intercambio de información únicamente entre usuarios que han compartido su clave y, se presupone, son de confianza.

La definición de roles de usuario sobre un *RCM* únicamente identifica dos tipos de participantes: quién es el usuario creador y qué usuarios tienen acceso para su edición. La política de privilegios definida únicamente restringe que el *RCM* puede ser eliminado por su usuario creador, no así por el resto. Esto implica que todos los usuarios con acceso a un *RCM* pueden llevar a cabo las mismas acciones de edición sin restricciones.

Mecanismos de sincronización entre el servidor y el cliente

Podemos separar el editor de *RCMs* en varios elementos que intervienen en la vista. Desde el punto de vista que nos afecta, distinguimos tres elementos cuya información es susceptible de mantenerse actualizada: el grafo, la lista de *References* y la lista de *Paths*. Un cambio de estado en cualquiera de ellos es necesario que desencadene una serie de peticiones entre el servidor y los clientes, de tal forma que todos los usuarios que se encuentren trabajando en ese momento con el *RCM* vean reflejados los cambios dinámicamente.

Se van a utilizar dos mecanismos distintos para sincronizar la información entre el servidor y los clientes.

Por un lado, el grafo va a apoyar en el *plug-in Channel* que facilita *Rappid*. *Channel* es una herramienta muy potente que ofrece la capacidad de colaboración en tiempo real a sus aplicaciones. El *plug-in* sincroniza los grafos de *JointJS* entre el servidor y los clientes y resuelve automáticamente cualquier conflicto que pudiera surgir entre ellos. Se limita únicamente a la sincronización del grafo, no así de las estructuras de datos auxiliares, como son las listas de referencias y de *Paths*.

La integración del *plug-in* en *RCMTool* es algo compleja. Sin embargo, en términos generales se trata de un servicio web que se publica de forma paralela en otro puerto del servidor Node.js y que atiende las peticiones que se producen desde el editor. Siempre que se produce un evento de cambio en el grafo de cualquiera de los clientes, el resto de editores se comunica con el servicio web para sincronizar el estado del grafo en tiempo real.

El *plug-in* proporciona una especie de *hub* o elemento centralizador que gestiona el conjunto de canales con los que trabaja el servidor. Siempre que se produce una petición, el servicio web comprueba si la instancia del grafo (también llamado canal) ya existe y está siendo consumida por algún usuario. Si existe, devuelve la instancia actualizada y sino la crea con el estado más reciente que se ha almacenado en la base de datos.

Por otra parte, la sincronización de las listas de *References* y *Paths* se basan en la utilización de *websockets*. Los *websockets* son una tecnología que permite una comunicación bidireccional entre cliente y servidor sobre un único *TCP*. En cierta manera, es un sustituto de *AJAX* como tecnología para obtener datos del servidor, ya que no es necesario pedirlos, es el propio servidor el que es capaz de enviarlos siempre que se actualizan.

Los *websockets* en *Node.js* se implementan a través del módulo *socket.io*, es el propio *socket* el que se instancia en el servidor para que se inicialice cuando se despliega. En el cliente también es necesario utilizar la librería *socket.io* para gestionar la comunicación con el servidor.

El funcionamiento, en este caso, es bastante sencillo. A pesar de ello, se requiere programación tanto de lado del cliente como del servidor.

En el lado del cliente es necesario instanciar el *socket*, indicando la *URL* a la que hay que notificar los cambios que se producen. Será necesario, por un lado, emitir notificaciones al *socket* del servidor siempre que se produzca un evento que desencadena cambios en el resto de clientes, y por otro, gestionar la recepción de notificaciones enviadas por el servidor cuando se produce un cambio en el resto de clientes.

Por su parte, en el lado del servidor se requiere gestionar la suscripción a eventos sobre cambios en cada uno de los *RCMs*. Cuando un usuario accede a un *RCM*, queda suscrito a los eventos que se producen en él. En el momento en que se produce una modificación sobre los elementos del *RCM*, el servidor debe ser capaz de procesar el evento y desencadenar una notificación en el resto de clientes para informarles del cambio producido.

Para llevar a cabo la gestión de accesos, *Node.js* se sirve de un contenedor que almacena todas las instancias de los *RCM* activos en el sistema, de una forma similar a la que utiliza el *plug-in Channel* de *Rappid*. Cada una de esas instancias, además, debe ser capaz de gestionar el conjunto de usuarios que están utilizando el *RCM* en ese momento. Cuando un usuario abandona el editor, el contenedor debe actualizarse para eliminar la suscripción a notificaciones del usuario sobre ese *RCM*.

En el caso de que un cliente pierda momentáneamente la conexión y siga modificando elementos del *RCM*, la herramienta informará al usuario de tal circunstancia y evitará que el usuario siga modificado el *RCM*. En el momento en que el usuario recupere la conexión, el estado del *RCM* quedará sincronizado con el estado del servidor, obviando los cambios que se puedan haber realizado en la copia en local.

Problemas surgidos y soluciones propuestas

A lo largo del proyecto han surgido una serie de inconvenientes que se han solventado sobre la marcha a medida que avanzaban las fases de desarrollo.

Los conflictos que han supuesto un mayor esfuerzo y dedicación para su subsanación han surgido de la necesidad de realizar peticiones en secuencia en el lado del servidor desde *Node.js* y de la adaptación del *plug-in Channel* de *Rappid* para su integración con la base de datos *MongoDB*.

En el primero de los casos, el objeto de realizar peticiones asíncronas desde *Node.js* viene motivado por las limitaciones que ofrece la *API* de *Mendeley* para la consulta de información. En concreto, la necesidad de la aplicación es consultar en un determinado momento el conjunto de referencias que contiene cada una de las carpetas asociadas al usuario en *Mendeley*. El gestor de referencias ofrece una serie de servicios que facilitan dicha información, pero no de la manera más conveniente.

Ha sido necesario consumir tres servicios diferentes para recuperar la información completa que requiere RCMTool:

- /folders

  Devuelve el listado de carpetas asociadas al usuario. De cada una de ellas, se recupera la siguiente información:

  [{ “id”: “”, “name”: “”, “created”: “”, “modified”: “”, “parent_id”: “”, “group_id”: “” }]

- /folders/{id}/documents

  Devuelve el listado de referencias asociadas a la carpeta que se pasa como {id}. De cada una de ellas, se recupera la siguiente información:

  [{ “id”: “” }]

- /documents/{id}

  Devuelve la información completa de la referencia que se pasa como {id}. Se recupera la siguiente información:

  [{ “id”: “”, “source”: “”, “type”: “”, “title”: “”, “authors”: “”, “abstract”: “”, … }]

Dado que *Mendeley* no ofrece la información requerida a través de una única llamada, surge la necesidad de realizar las operaciones de acceso a datos en

secuencia. En un primer bucle, es necesario recorrer cada una de las carpetas asociadas al usuario. A su vez, de cada una de ellas es necesario recuperar la información de sus referencias y, en un tercer bucle interno, recuperar la información completa de cada una de esas referencias.

Node.js por defecto gestiona las peticiones de manera asíncrona, por lo que se ha utilizado uno de sus módulos para simplificar la labor de programación simulando peticiones síncronas. La librería *async* permite resolver un conjunto de peticiones de forma secuencial. Este comportamiento lo consigue encadenando funciones *callback* a la llamada anterior, y así sucesivamente.

Por otra parte, el problema de sincronización entre el *plug-in Channel* y la base de datos *MongoDB* se entiende desde el punto de vista de que el *plug-in* se sirve de un servicio web interno, paralelo a la ejecución del servidor de *Node.js*. Su metodología de trabajo interna consiste en utilizar un *array* para gestionar cada uno de los *RCMs* que hay activos en la aplicación. Cuando un usuario accede a un *RCM*, el servicio web comprueba si existe una instancia creada, es decir, si algún otro usuario está trabajando en ese momento con el RCM. En caso afirmativo, devuelve la misma instancia de trabajo al nuevo usuario. En caso contrario, crea una nueva. De esta forma, cualquier cambio realizado sobre el grafo se ve reflejado sobre el servidor y, posteriormente, sobre el resto de usuarios.

Sin embargo, surge un gran problema que es necesario subsanar. Dado que *Channel* mantiene un *array* interno de instancias, siempre que se reinicia el servidor *Node.js*, este *array* se borra por completo. En consecuencia, cuando un usuario accede de nuevo a la aplicación e intenta acceder a un *RCM* existente, la instancia no existe y crea una nueva, con el grafo vacío.

Ha sido necesario modificar el *script* en el servidor para que el servicio web tenga en cuenta que, si no existe la instancia, debe comprobar en la base de datos si existe el *RCM* y, en ese caso, cargar sobre la instancia al vuelo el objeto *SVG* almacenado en *MongoDB*. Si, por el contrario, tampoco existe el RCM en la capa de persistencia, entonces sí debe generar un nuevo grafo con su correspondiente instancia vacía.

Resultados obtenidos

La reestructuración interna de la herramienta aporta una base mucho más estable y flexible a cambios. La separación entre capas y su separación en módulos internos facilita las labores de programación de cara a futuros cambios.

Técnicamente, contar con las últimas versiones de cada tecnología supone contar con funcionalidades específicas avanzadas, uso de *APIs* de desarrollo con muchas más posibilidades y la corrección de posibles errores en las fuentes originales.

La mejora de errores aporta trazabilidad interna a la hora de ofrecer soporte de la herramienta y usabilidad de cara al usuario final que es informado en todo momento de los problemas surgidos en la herramienta.

En cuanto a las funcionalidades avanzadas programadas, suponen un avance de cara a la puesta en producción de una herramienta cuyo uso era limitado y era necesario pulir.

El rediseño gráfico es quizás la única labor donde los cambios son más cualitativos, más difíciles de medir. En cualquier caso, facilitan en gran medida la interacción del usuario con la herramienta y le presentan la información de una manera más clara y concisa.

La gestión de referencias es un paso lógico de la herramienta. Se hacía necesario revisar la estructura interna de almacenamiento para asemejarla, en la medida de lo posible, a la gestión que se lleva a cabo en los gestores de referencias modernos. La creación del concepto de carpeta permite administrar de una manera mucho más eficiente las referencias. De cara al usuario, los cambios en el manejo de referencias simplifican bastante su labor. Además, con estos cambios se facilita la reutilización de información, con lo que se almacenan los datos de una manera mucho más eficiente.

La definición de nuevos modelos de datos es también un paso lógico como consecuencia de la reestructuración. Supone la utilización de modelos de trabajo

definidos, pero a su vez mucho más permisivos a cambios. También simplifican el desarrollo de cara a la realización de operaciones en base de datos.

La integración con *Mendeley* permite la importación de referencias sin necesidad de cargarlas una a una en el sistema. Además, ejemplifica los procesos de integración de cara a futuras integraciones con gestores de referencias similares.

La edición colaborativa fomenta el trabajo en grupo de manera concurrente. Es quizás la funcionalidad que más valor aporta a la herramienta porque ofrece la posibilidad de compartir información con el resto de usuarios del sistema. Es bastante útil porque pone a disposición de los interesados cierta información sin necesidad de buscar otros mecanismos de comunicación, reduciendo costes, aumentando la transparencia y de una forma segura.

Conclusiones

En líneas generales, los puntos de actuación del TFM se han orientado a cubrir las necesidades más básicas de clientes potenciales que pueda tener la herramienta a corto-medio plazo. Sin embargo, quedan abiertas multitud de posibilidades de cara a futuras ampliaciones.

Como toma de contacto con el uso de herramientas gráficas para el tratamiento de problemas lingüísticos permite hacerse a la idea de la complejidad con la que cuenta una herramienta de procesamiento de lenguaje. Más allá de la dificultad de los procesos propios del tratamiento de la información, surge la necesidad de adaptar todos esos requisitos a métodos funcionales reutilizables que faciliten las labores de mantenimiento.

Desde el punto de vista técnico, que sin duda es en el que más me he involucrado, supone una constante búsqueda de documentación relacionada con las tecnologías que se utilizan y sus posibles integraciones. El trabajo con *Node.js*, que prácticamente es el entorno de ejecución en torno al que giran el resto de tecnologías, supone un cambio total en la forma de pensar del desarrollador. Te obliga a valorar constantemente qué parte de la lógica negocio se debe gestionar desde la parte servidor y cuál desde el cliente, hasta qué punto separar una capa de otra, qué módulos puedes integrar para reutilizar información y, sobre todo, cómo plasmar todas las ideas de trabajo que tienes en tu cabeza en un entorno de desarrollo asíncrono. Por decirlo de alguna forma, hasta hora te han mostrado la parte teórica que te da pinceladas de la infinidad de posibilidades que tienes al enfrentarte a un problema, pero es ahora cuando realmente tienes que demostrar que has asimilado todos esos conceptos y los has sabido organizar, de alguna forma, para hacer frente a un proyecto real de trabajo.

Personalmente, el reto de enfrentarse al proyecto me ha dejado un buen sabor de boca en cuanto al conocimiento global adquirido. Muchas veces no es necesario centrarse en una sola idea de trabajo, sino conocer un poquito de todo y saber qué elegir de cada una de ellas en función de tus necesidades. Esta es un poco la filosofía del máster, no especializarse en un tema concreto y cerrado,

sino abarcar un curso multidisciplinar que ofrece muchos puntos de vista que te permiten abrir la mente y enfrentarte a un mismo problema desde distintas perspectivas.

El resultado creo que es el esperado y se adecúa bastante a lo que se pretendía al inicio del proyecto. Hay una serie de aspectos con los que me gustaría haber trabajado y que quedan pendientes para futuros desarrolladores, como la posibilidad de automatizar el despliegue de la aplicación. Sin embargo, considero que las labores llevadas a cabo han sido lo suficientemente complejas y las he podido orientar hacia donde se me había propuesto al inicio del TFM.

A pesar de ello, entiendo que de cara a un usuario externo puede parecer que la labor realizada es menor de lo que en realidad ha sido. Por poner un ejemplo, el proceso de reestructuración ha sido bastante complejo y, sin embargo, de cara al usuario es prácticamente transparente, apenas va a notar mejoras funcionales. Considero que gran parte del trabajo positivo se va a valorar de cara a futuras implementaciones, no tanto ahora mismo en cuanto a un desarrollo actual.

Posibilidades de futuro

La remodelación de la herramienta tiene como objetivo facilitar la implementación de nuevas funcionalidades a futuros desarrolladores que colaboren en el proyecto académico:

1. Procesamiento del lenguaje

   De cara a la mejora de resultados en la fase de procesamiento de definiciones, se podrían implementar reglas obtenidas a través de un conjunto de datos de entrenamiento basado en definiciones.

2. Métricas

   Uso de métricas como objetivo para obtener información implícita en las definiciones de forma visual y aportar una base matemática que dé fiabilidad a las conclusiones.

3. Gestores de referencias

   Integrar la herramienta con otros gestores que permitan aumentar las fuentes de referencias. Se pueden utilizar bases sentadas con *Mendeley*.

4. Plataformas de *RCMs*

   Creación de buscadores de definiciones. Enlazar *RCMs* mediante la expansión de los conceptos que lo conforman enriqueciendo así la literatura.

5. Instrumentos de distribución

   Automatizar el despliegue de aplicaciones dentro de contenedores de software, de tal forma que el usuario de abstraiga por completo del proceso de instalación y configuración inicial de la herramienta.

6. Roles y autorizaciones

   Definir nuevos permisos de participación en un *RCM* una vez compartido. Facilitar mecanismos para la autorización de privilegios entre usuarios de un mismo *RCM*.

Bibliografía

[1] Adán, C. S. (2014). Herramienta para el desarrollo de References-enriched Concept Maps.

[2] Rodriguez-Priego, E. (2013). References-enriched Concept Map: a tool for collecting and comparing disparate definitions appearing in multiple references. *Journal of Information Science*.

[3] Adán, C. S. (2014). RCMTool. *Aplicación Software*.

[4] *Node.js.* (s.f.). Obtenido de Node.js: https://nodejs.org/docs/latest-v4.x/api/

[5] *Express.js.* (s.f.). Obtenido de Express.js: http://expressjs.com/es/4x/api.html

[6] *MongoDB.* (s.f.). Obtenido de MongoDB: https://docs.mongodb.com/

[7] *Hogan.js.* (s.f.). Obtenido de Hogan.js: http://twitter.github.io/hogan.js/

[8] *jQuery.* (s.f.). Obtenido de jQuery: http://api.jquery.com/

[9] *Rappid.* (s.f.). Obtenido de Rappid: http://resources.jointjs.com/docs/rappid/v2.0/index.html

[10] *BibTeX.* (s.f.). Obtenido de BibTeX: http://www.bibtex.org/Format/

[11] *Mendeley.* (s.f.). Obtenido de Mendeley: https://api.mendeley.com/apidocs/docs

[12] *Oauth*. (s.f.). Obtenido de Oauth: https://oauth.net/

[13] *Migración Express 3 a 4.* (s.f.). Obtenido de Migración Express 3 a 4: http://expressjs.com/es/guide/migrating-4.html

[14] *UnderScore.* (s.f.). Obtenido de UnderScore: http://underscorejs.org/

[15] *Object-hash.* (s.f.). Obtenido de Object-hash: https://github.com/puleos/object-hash

[16] *Crypto.* (s.f.). Obtenido de Crypto: https://nodejs.org/api/crypto.html

[17] *jQueryValidation.* (s.f.). Obtenido de jQueryValidation: https://jqueryvalidation.org/